Pinball Project

Place a disc of radius r at the corners of an equilateral triangle of side s, where r is less than s/2. Our ball is going to be a point particle, for mathematical simplicity. The game is to shoot the particle with unit speed in some direction from a specified point, and see how long it takes to escape; the particle will bounce around the discs some number of times before escaping, depending on the initial position and direction. We assume that the reflections are perfectly elastic, so no energy is lost and the speed remains 1 throughout.

This is an example of what is called a *chaotic* dynamical system: very small changes in the initial position or direction of the ball can produce a trajectory that soon looks nothing like the original trajectory. Each bounce causes small differences to be magnified, so that after a few bounces the originally close trajectories are bouncing off different balls, or one is missing the balls altogether while the other is still captured.

We are not going to study the theory of such systems; we are just going to simulate this one with the computer, as an exercise in the geometry of reflections which we studied in chapter 0, and in intersecting lines in parametric form with circles.

Here is the algebra involved for finding the intersection. Suppose a particle is moving with constant velocity **v**, having started at time zero at point **x**. Then its location at time t is **y** = **x** + t**v**, which is a parametric form of a line as you recognize. We want to find when (and if) the particle hits the circle with center **c** and radius r, and we are only interested in solutions for *positive* t (going forward in time), and we are only interested in cases where the particle starts *outside* the circle (we don't want to start inside the circle and bounce around inside the circle). The condition for the line intersecting the circle at time t is $(\mathbf{x}+\mathbf{v}t-\mathbf{c})\bullet(\mathbf{x}+\mathbf{v}t-\mathbf{c}) = r^2$, which says that the distance from the point on the line to the center of the circle is r. *Let's assume from now on* that $\mathbf{v}\bullet\mathbf{v}=1$ (that is, the speed is 1) to make the algebra look a little simpler. Using the algebra of the dot product, you get this equation for t: $t^2 - 2t\mathbf{v}\bullet(\mathbf{c}-\mathbf{x}) + (\mathbf{c}-\mathbf{x})\bullet(\mathbf{c}-\mathbf{x}) - r^2 = 0$. As every Georgia Tech student knows (but I once saw a classroom interview at UGA where the quadratic formula was incorrectly written!), this has solutions

$$t = \mathbf{v}\bullet(\mathbf{c}-\mathbf{x}) \pm \sqrt{D}, \text{ where } D = (\mathbf{v}\bullet(\mathbf{c}-\mathbf{x}))^2 - (\mathbf{c}-\mathbf{x})\bullet(\mathbf{c}-\mathbf{x}) + r^2.$$

Since we are assuming that $\mathbf{x}_0$ is *outside* the circle, the term $-(\mathbf{c}-\mathbf{x})\bullet(\mathbf{c}-\mathbf{x}) + r^2$ is negative. So we come to the following conclusions:

(1) If $D < 0$, there is no intersection of the line with the circle (because there is no real solution for t).

(2) If $D = 0$, the line is tangent to the circle (there is just one intersection point). With probability one this is not going to happen, and even if it did, the particle would just continue on the straight line anyway. So this is just like (1).

(3) If $D > 0$ and $\mathbf{v}\bullet(\mathbf{c}-\mathbf{x}) < 0$, then both solutions for t are negative (because $\sqrt{D} < |\mathbf{v}\bullet(\mathbf{c}-\mathbf{x})|$ from our observation above about **x** being outside the circle). So the intersections of the line with the circle are behind us in time, and we are not interested.

(4) If $D > 0$ and $\mathbf{v}\bullet(\mathbf{c}-\mathbf{x}) > 0$, then both solutions for t are positive. We want the first time that the line hits the circle, so we take the smaller solution.

As good computer programmers, you are always interested in efficiency, right? So if you have a function t = intersect(**c**, r, **x**, **v**) whose job is to return the smallest positive t for which the particle with velocity **v** (assumed to be a unit vector) and initial location **x** hits the circle with center **c** and radius r, and you are willing to assume that you will only call the function with the initial point outside the circle, then you might do it in this order:
(i) Compute $\mathbf{v}\bullet(\mathbf{c}-\mathbf{x})$. If this is less than or equal zero, return no solution.
(ii) Compute D (using the value of $\mathbf{v}\bullet(\mathbf{c}-\mathbf{x})$ that you already computed to save work). If D ≤ 0, return no solution.
(iii) If D > 0, return the solution $\mathbf{v}\bullet(\mathbf{c}-\mathbf{x})-\sqrt{D}$, as this will be the smallest positive solution.

This way, we exit sometimes with just a single dot product computed. And we only compute a square root when there really is a solution. Of course if you lied and had the initial point inside the circle, this could return a negative t, so this would only be a good idea if you intend to use this function only in this application!

The algebra for finding the reflection of the particle off the circle at the point of intersection is just like what we did in chapter 0. Suppose the particle with velocity **v** hits the circle with center **c** at point **x**. The simple geometric fact to notice is that the vector **c** - **x** from **x** to the center of the circle is perpendicular to the tangent to the circle at point **x**. So reflecting the particle off this tangent using the methods from chapter 0, we get for the velocity vector **w** of the reflected particle

$$\mathbf{w}=\mathbf{v}-\left\{\frac{2\mathbf{v}\bullet(\mathbf{c}-\mathbf{x})}{(\mathbf{c}-\mathbf{x})\bullet(\mathbf{c}-\mathbf{x})}\right\}(\mathbf{c}-\mathbf{x})$$

A moment's "reflection" (sorry for the pun) will convince you that **w** has the same length as **v**, it is just pointing in a different direction.

Note how simple this is, nothing but rational arithmetic. Perhaps if you had decided to work out the reflection before having read chapter 0, you might have tried to compute some angles or something, with the idea that the angle of reflection equals angle of incidence. It is much better to use the vector algebra methods!

So, here is the plan. To keep things simple, let's put the centers of our circles at the points $(s/2,\,-s\sqrt{3}/6),\ (-s/2,\,-s\sqrt{3}/6),\ \mathit{and}\ (0,\,s\sqrt{3}/3)$. These are the centers of an equilateral triangle with side s and centroid at the origin. Let's always start the particle off from the origin, with unit speed, so the only choice we have is the choice of direction. As we said, the radius of our circles will be required to be less than s/2, so the particle can escape (we are not closed off by the circles).

You will write two functions t = intersect(**c**, r, **x**, **v**) and **w** = reflect(**c**, **x**, **v**) as described above that do the work Start the particle off from the origin in some direction with unit speed (how to do this will be described in a moment). So $\mathbf{x}_0$ = (0, 0) and $\mathbf{v}_0$ = whatever, as long as it has length 1. Call the function t = intersect(**c**, r, $\mathbf{x}_0$, $\mathbf{v}_0$) using the three centers $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$ in succession, to see if you get an intersection with one of the circles (you could indicate that you got no intersection in any way you want, such as returning –1 if no intersection, or have another variable that is a flag if that method offends you). If you do get an intersection, let t_0 be the value returned. Then let $\mathbf{x}_1 = \mathbf{x}_0 + t_0\mathbf{v}_0$, which will be a point on the circle **c** it hits (this is one of the three circles $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$). Now call $\mathbf{v}_1$ = reflect(**c**, $\mathbf{x}_1$, $\mathbf{v}_0$). Now you are at the point $\mathbf{x}_1$ traveling away from the circle you just hit, now with velocity $\mathbf{v}_1$. So you are ready to repeat! Except unlike the initial call, you will only check two circles for intersection. Do not check the circle you are sitting on; that won't get you anywhere and also might cause problems with your program since intesect() might assume that you are starting outside the circle in question. So in your loop you will also need to keep track of what circle you hit last, so as to not check for that one.

Do you need to try to intersect both of the other circles when going away from a circle? If the circles have small enough radius, you can see that you can't possibly hit more than one other circle going away from a circle, but if the circles have radius near the maximum of s/2, then it is possible to hit both other circles. To keep it simple, let's just check both other circles, and if you happen to get a solution for both, take the one with the smallest t, of course, because you will reflect off of that one.

In this way you will produce a sequence of points and velocities $\mathbf{x}_n = \mathbf{x}_{n-1} + t_{n-1}\mathbf{v}_{n-1}$ and $\mathbf{v}_n$ = reflect(c, $\mathbf{x}_n$, $\mathbf{v}_{n-1}$), until you get no intersection. Count how many hits you get before you escape from the circles.

What initial velocities should you use? Lets make the game to find the most hits that you can before escaping. Choosing an initial velocity with unit speed amounts to choosing an initial angle θ, and then letting **v** = (cos θ, sin θ).

Method 1: randomly choose a lot of different angles θ between 0 and 2π using a random number generator. How many you can try in a reasonable time will depend on how fast your code runs. My program did billions in just a couple of seconds for the case s = 6 and r = 1, but my program was in c and your java program that you will probably write will probably be slower (of course you can use any language you like).
Method 2: choose a lot of angles equally spaced between 0 and 2π.
Method 3: make inspired guesses, if you think you have an idea how to get a lot of hits before escaping. Good luck.
Method 4: take a course in the physics department and learn some tricks for coming very close to a long periodic orbit (none of those start at the origin, so you won't actually get infinitely many hits no matter what you try). OK, just kidding.

Let's do it with just these two choices for machine setup: s = 6 and r = 1, and then s = 6 and r = 2. Keep a frequency count of how many of your orbits got 0 hits, how many got 1 hit, 2 hits, ...etc before escaping. Here's how to do that: make an array freq[1000] which you initialize to zeroes, and each time you get 3 hits, for example, you will increment freq[3], so at the end, freq[3] will contain the number of orbits that yielded exactly three hits before escaping. Don't worry, you will not get anywhere near 1000 hits;

I just now ran 100 million randomly chosen angles with s = 6 and r = 1 and the most hits I got before escaping was 13. You can do better than that with more trials, and with r = 2 you will get more hits of course. In any case you can have your program check to see if the number of hits exceeds your freq array size if you are worried about it crashing.

What to Report

So here is what you can report:
(1) The relative frequency (that is, frequency divided by number of trials) of num hits you got in the two cases mentioned, and how many trials it was based on (You will notice that the frequency drops off geometrically fast with the number of hits). Do that both for the random method and the systematic method of choosing the initial angle. For the same number of trials, which method came up with the longest orbit, the random way or the systematic way?
(2) For the top 10 number of hits you got, give the initial angle and give the sequence of circles that were hit in the orbit before escaping. If you found your best orbit by some other method than just brute force as described above, tell us how you did it.
(3) If you are quick at making graphics displays, make a display that shows the orbit for a given starting angle as a ball dynamically moving around from one circle to the next, and use this to display some of your orbits.
(3) might be too time-consuming, so it is not required for the project.

Report your longest orbit for the case r = 1, s = 6. Let's see who is the winner for that case!

(Programming considerations: (1) Be sure to use double-precision arithmetic in your calculations to minimize the damage of roundoff error. (2) Be sure to step through your code in a debugger for a few bounces to verify that it seems to really be bouncing off the circles as you expect; it is easy to make little mistakes and you should always step through your code in a debugger as a sanity check. If you do the visual display of the ball bouncing around, that would make it easy to see if it is making sense!)